Proceedings of the 2022 Winter Simulation Conference
B. Feng, G. Pedrielli, Y. Peng, S. Shashaani, E. Song, C.G. Corlu, L.H. Lee, E.P. Chew, T. Roeder, and P. Lendermann, eds.

TOWARDS A UNIFYING FRAMEWORK FOR MODELING, EXECUTION, SIMULATION, AND OPTIMIZATION OF RESOURCE-AWARE BUSINESS PROCESSES

Asvin Goel

Kühne Logistics University
Großer Grasbrook 17
20457 Hamburg, GERMANY

ABSTRACT

This paper proposes an extension to BPMN 2.0 to be used within a framework allowing to execute and simulate resource-aware business processes. The framework consists of three core components: a data provider responsible for acquiring all relevant information, an execution engine responsible for advancing process execution according to the respective execution logic, and a controller responsible for making all decisions required during process execution. The data provider and controller can be easily replaced depending on the use case and the entire execution logic is encapsulated within the execution engine. The framework is designed in such a way that it allows any decision mechanism to be deployed ranging from manual decision making to sophisticated optimization algorithms.

1 INTRODUCTION AND RELATED WORK

In order to improve the performance of business operations many companies apply methods and tools developed in the fields of *Business Process Management*, *Discrete-Event System Simulation*, and *Optimization*. From the field of *Business Process Management*, the so-called *Business Process Model and Notation (BPMN 2.0)* (Object Management Group 2013) has become a standard in modelling business processes using a graphical notation that is easy to learn and use. Unfortunately, BPMN 2.0 has severe limitations concerning the consideration of resource requirements. While BPMN 2.0 allows, to some extend, to specify resources that are needed for subprocesses and tasks, there is no way of ensuring that the required resources are actually available when executing business processes. In *Discrete-Event System Simulation*, resources play a central role and are usually modelled as entities that can accept a limited number of work items. Resources are usually equipped with queues that can hold work items which have to wait until the resource becomes available to accept the work item. Although graphical notations used for *Discrete-Event System Simulation* share some similarities with BPMN 2.0, a translation from one notation to another is often not straight-forward. Both BPMN 2.0 and *Discrete-Event System Simulation* are not designed to find decisions that contribute to the optimization of performance goals, in particular, regarding the assignment and allocation of resources. In the field of *Optimization*, a multitude of models and algorithms have been developed allowing to optimize the assignment and allocation of resources. These models are usually based on *Mixed-Integer Programming* and efficient algorithms are often tailor-made for the respective problem. Although, such optimization techniques can bring substantial performance improvements, developing models and algorithms is often a prohibitively expensive task preventing widespread utilization.

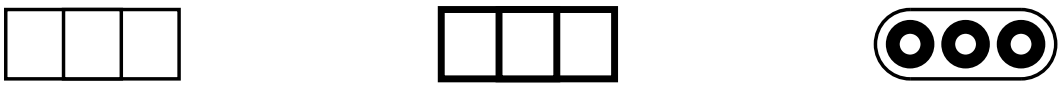
The potential of enriching BPMN 2.0 in such a way that it allows for *Discrete-Event System Simulation* has been identified early and several approaches have been proposed. With the *Business Process Simulation Specification (BPSim)* (Workflow Management Coalition 2013), for example, it

is possible to specify simulation-relevant data such as resource roles, quantities, and availabilities. Pufahl et al. (2017) propose an architecture of a BPMN process simulator which maps BPMN constructs into a Discrete-Event Simulation model. Onggo et al. (2018) propose to extend BPMN 2.0 introducing the concept of *Shared Tasks* which are used to indicate tasks that share the same resource with a limited capacity. Pufahl and Weske (2013) propose an execution semantics for so-called *Batch Activities* allowing to dynamically generate batches across multiple process instances. Wagner (2020) proposed so-called *Resource-constrained Activities* which are visually represented through markers on incoming sequence flows and Wagner (2021) discusses the use of so-called *Processing Activities* which are a resource-constrained activities that are performed at a processing station. In all of these works, resources are assumed to be entities with a predefined set of characteristics which are required for process execution. In contrast to other work, this paper takes a novel view on resources by modelling them through so-called *Resource Activities* comprising several subprocesses describing the characteristics and behaviour of the resources. As these subprocesses can be arbitrarily provided by the modeller, the proposal has a high flexibility with respect to the type of resources that can be modelled. *Resource Activities* can be included in business process models just like any other activities and the allocation of resources can be viewed as the initiation of a collaboration between different processes subject to a specific choreography. *Request Activities* and *Release Activities*, as proposed by Goel and Lin (2022), allow an allocation of resources similar to the *Seize-Delay-Release* modelling pattern used in GPSS (Gordon 1961).

This paper seeks at contributing towards a unifying framework for modelling resource-aware business processes that is capable of executing and simulating these processes and optimizing resource allocations using different approaches ranging from manual decisions to sophisticated optimization algorithms. After presenting an extension for BPMN 2.0 allowing to model resources and resource requirements, this paper proposes a flexible architecture of the framework based on the observer design pattern and presents an example use case demonstrating the applicability.

2 RESOURCE-AWARE BUSINESS PROCESSES

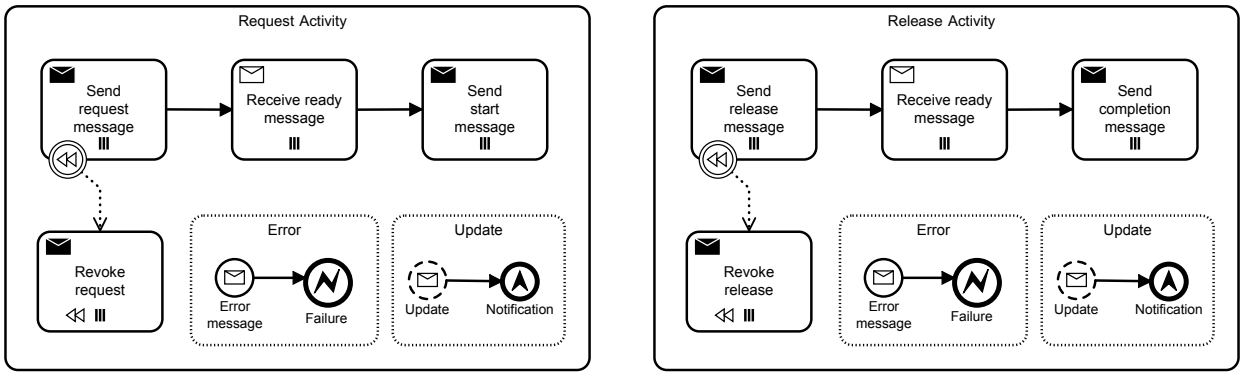
In order to model resource requirements in business process models, Goel and Lin (2022) proposed to extend BPMN 2.0 by introducing so-called *Request Activities* and *Release Activities*. *Request Activities* indicate that one or more resources are requested and must be allocated to the requesting process before it can continue, whereas *Release Activities* indicate that one or more resources previously requested are no longer required. These modelling elements are represented by three connected squares representing a horizontal stack or a queue as shown in Figures 1a and 1b. *Request Activities* are drawn with normal line width and *Release Activities* are drawn with thick line width.



(a) *Request Activity* (b) *Release Activity* (c) *Resource Activity*

Figure 1: Additional modelling elements for resource-aware business processes.

The behaviour of *Request Activities* and *Release Activities* is illustrated by the subprocesses shown in Figures 2a and 2b. A *Request Activity* can simultaneously request one or multiple resources. It sends a request message to each resource allocated, waits until all resources are ready to be used and have sent a ready message, and then, sends a start message to all allocated resources. If any of the resources sends an error message, the *Request Activity* terminates with a failure. Before termination, however, a message is sent to all allocated resources that the respective request is revoked. Similarly, if the *Request Activity* is interrupted, e.g. by an interrupting event attached to the boundary, all



(a) *Request Activity* (b) *Release Activity*

Figure 2: Behaviour of *Request Activities* and *Release Activities* (Goel and Lin, 2022).

requests are revoked. A *Release Activity* behaves analogously, but sends a release message to all resources that are no longer needed.

Goel and Lin (2022) focussed on representing resource requirements and did not detail the modelling of resources except for the requirement, that they must be able to adequately exchange messages with *Request Activities* and *Release Activities*. In order to facilitate an automated allocation of resources to requests, the allocation mechanism needs to be able to identify resource models and their characteristics. This requires to include additional information. Before showing how this information can be provided, we first have to note that resources relate to processes in different ways. On the one hand, resources can be used by other processes or can provide services requested by other processes. On the other hand, resources may have to conduct their own subprocesses and can be provided by and within business processes. Furthermore, resources may require the support of other resources in order to provide a service to a requesting process. For example, a truck requires a trailer to be able to transport shipments. To support a variety of resources with different characteristics, this paper proposes to model resources as dedicated modelling elements to be used within business process models. These modelling elements are named *Resource Activities* and a stylized conveyor belt as shown in Figure 1c is proposed as graphical representation. A *Resource Activity* can be included within the sequence flow of any process model just like any other activity.

The behaviour of *Resource Activities* is illustrated by the subprocess shown in Figure 3. Each *Resource Activity* comprises multiple subprocesses that can be specified by the modeller: a *Default* subprocess, a *Prepare* subprocess, a *Service* subprocess, and a *Finish* subprocess. When a *Resource Activity* receives a token from a process execution engine, the *Default* subprocess is initiated. As long as the *Default* subprocess is alive, the resource can be allocated to requests. Note, that the mechanism to allocate resources to requests is not specified in the model. Later in this paper, we will discuss how resources can be allocated to requests. When a resource is allocated to a request, a request message is sent from the respective *Request Activity* describing the job that is to be performed by the resource. The *Prepare* subprocess is executed by the resource to ensure that the resource can actually fulfil the requirements as requested. After completion of the *Prepare* subprocess, a ready message is sent to the respective *Request Activity*. The *Request Activity* responds with a start message, when all requested resources are ready. Should the request be revoked before the start message is received, the resource immediately initiates the *Finish* subprocess described later. Otherwise, the *Service* subprocess is conducted which can implement any predefined choreography with the requesting process. Simultaneously, the *Resource Activity* awaits a release message. When the *Service* subprocess is completed and the release message is obtained, a ready message is sent to the respective *Release Activity*. The *Release Activity* responds with a completion message, when all

allocated resources have completed the *Service* subprocess. Thereafter, the resource conducts the *Finish* subprocess. Note that, if any of these support subprocesses of a *Resource Activity* are not needed, they can be assumed to be empty processes that immediately terminate after being invoked. Should an uncaught error occur during execution of any of these support subprocesses, the *Resource Activity* terminates with an error. Before termination, however, the *Failure* event-subprocess is executed, sending an error message to all processes that requested the resource.

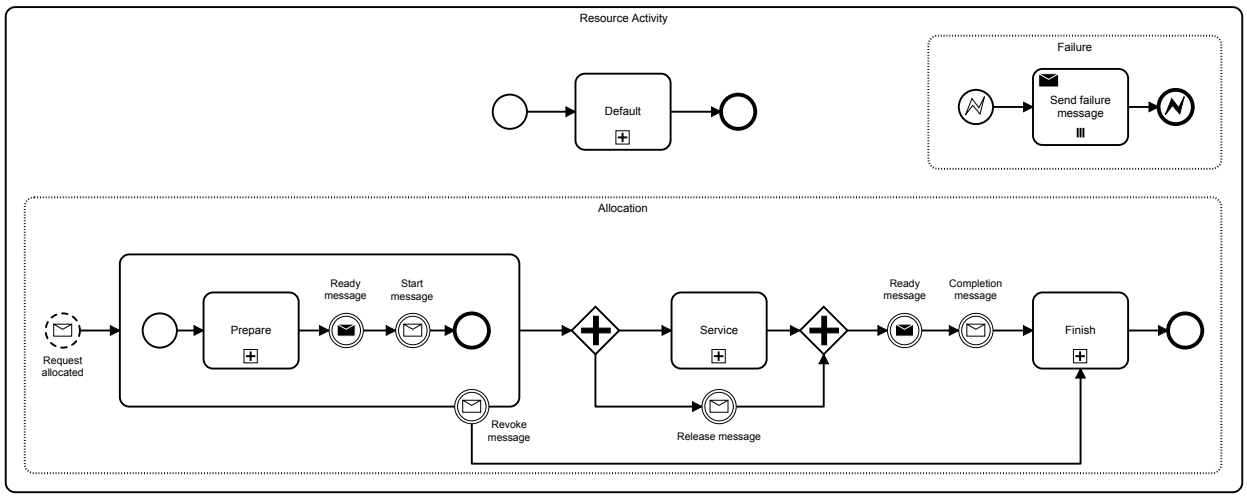


Figure 3: Behaviour of *Resource Activities*.

To allow for automatic execution of process instances, we will use the notion of tokens moving through the process models to represent the state of the system. For each token a set of status attributes is available. Throughout process execution, the values of these attributes can be modified using operators and the set of permissible values can be constrained by restrictions.

Each token is associated with a set of status attributes which can be declared within process models using the BPMN 2.0 extension mechanism as shown below.

```
<extensionElements>
  <execution:status>
    <execution:attribute name="some_attribute" type="xs:string" />
    <execution:attribute name="other_attribute" type="xs:decimal" objective="maximize" weight="1"/>
  </execution:status>
</extensionElements>
```

Within an `<execution:status>` block, any number of status attributes can be provided. Each `<execution:attribute>` element requires the attributes `name` and `type` to be provided. The `name` attribute defines a name that must be unique within the scope of the token and that can be used to refer to the attribute during execution of a particular process instance. The `type` attribute defines the data type and can be set to `"xs:string"`, `"xs:integer"`, `"xs:decimal"`, or `"xs:boolean"`.

The `objective` attribute is optional and indicates whether the status attribute contributes to a global objective which is to maximize all status attributes with `objective="maximize"` and minimize all status attributes with `objective="minimize"`. The tradeoff between different objectives can be indicated by specifying different `weight` attributes to set the multiplier in a weighted sum of objectives.

The values of status attributes can be constrained by restrictions added to process models as shown below.

```
<extensionElements>
  <execution:restrictions>
    <execution:restriction attribute="some_attribute_name" negate="true">
```

```
      <execution:enumeration value="Apple"/>
      <execution:enumeration value="Orange"/>
    </execution:restriction>
    <execution:restriction attribute="another_attribute_name" required="true">
      <execution:minInclusive value="0"/>
      <execution:maxInclusive value="100"/>
    </execution:restriction>
  </execution:restrictions>
</extensionElements>
```

Within an `<execution:restrictions>` block, any number of restrictions can be provided. Each `<execution:restriction>` element requires an attribute `attribute` indicating the name of the status attribute to be restricted. A restriction may have the optional boolean attribute `required` indicating whether the given status attribute must be defined. Furthermore, restriction may have the optional boolean attribute `negate` indicating whether the outcome of the restriction is to be negated. If both `required` and `negate` are `"true"`, the given status attribute must not be defined. Furthermore, any number of `<execution:enumeration>` elements can be added to a restriction each giving a `value` that the given status attribute is allowed to take (or must not take in case `negate="true"`). The given status attribute can be constrained to a value larger or equal to a `value` given in a `<execution:minInclusive>` element or to a value smaller or equal to a `value` given in a `<execution:maxInclusive>` element. A negation allows to model strictly larger than or smaller than restrictions.

At every node in a process model, the values of status attributes can be modified by operators added to the node as shown below.

```
<extensionElements>
  <execution:operators>
    <execution:operator name="expression" attribute="volume">
      <execution:parameter name="expression" value="height*width*length" />
    </execution:operator>
    <execution:operator name="lookup" attribute="distance">
      <execution:parameter name="source" value="distanceTable" />
      <execution:parameter name="origin" value="current_location" />
      <execution:parameter name="destination" value="customer_location" />
    </execution:operator>
    <execution:operator name="random" attribute="duration">
      <execution:parameter name="distribution" value="exponential" />
      <execution:parameter name="lambda" value="rate" />
    </execution:operator>
    <execution:operator name="choice" attribute="x" />
  </execution:operators>
</extensionElements>
```

Within an `<execution:operators>` block, any number of operators to be executed in the given order can be provided. Each `<execution:operator>` element requires an attribute `attribute` indicating the name of the status attribute in which the result obtained by applying the operator is stored. Furthermore, an attribute `name` indicating the name of the operator to be applied. The operator name can be `"expression"`, `"lookup"`, `"random"`, or `"choice"`. A list of `<execution:parameter>` elements with `name` and `value` attributes can be provided for each operator. Additional operators using the same parameter definitions can be implemented by custom tools.

The `"expression"` operator expects an element `<execution:parameter>` with attribute `name="expression"` and `value` being the expression to be applied. The example above determines the value of the attribute with name `"volume"` by multiplying the attribute values of the attributes with names `"height"`, `"width"`, and `"length"`. The `"lookup"` operator expects an element `<execution:parameter>` with attribute `name="source"` and `value` being the name of the lookup function. For each required input parameter of the lookup function, an `<execution:parameter>` element with attribute `name` indicating the name

of an input parameter and attribute `value` indicating the respective name of the status attribute to be used as parameter are required. The example above determines the value of the attribute with name `"distance"` using a lookup function named `"distanceTable"` that expects the parameters `"origin"` and `"destination"` which are stored in the status attributes named `"current_location"` and `"customer_location"`. Custom tools implementing the framework can provide any kind of lookup function required. The `"random"` operator expects a parameter with name `distribution`, and other distribution-dependent parameters. The example above determines the value of the attribute with name `"duration"` using an exponential distribution with the parameter named `"lambda"` that is set to value of the attributes with name `"rate"`. The set of random operators can be arbitrarily extended by custom implementations. The `"choice"` operator requires a decision to be made. In the example above, the decision made is stored in the attribute with name `"x"`. The set of possible choices that can be taken is determined by the the data type of the respective attribute and can be further constrained by adding respective restrictions.

For every *Resource Activity* a list of key-values pairs must be provided specifying the information required from any request to which the resource is allocated as shown below.

```
<extensionElements>
  <execution:job>
    <execution:content key="a_key" attribute="an_attribute_name" />
    <execution:content key="another_key" attribute="another_attribute_name" />
  </execution:job>
</extensionElements>
```

Within an `<execution:job>` block, any number of `<execution:content>` elements can be given, providing a dictionary translating each `key` of the job description given with a request message into the `attribute` of the token status of the respective *Allocation* event-subprocess shown in Figures 3.

For each *Request Activity* a list of requested resource allocations can be provided as shown below.

```
<extensionElements>
  <execution:allocations>
    <execution:request id="some_id">
      <execution:job>
        <execution:content key="a_key" attribute="an_attribute_name" />
        <execution:content key="another_key" value="a_value" />
      </execution:job>
    </execution:request>
  </execution:allocations>
</extensionElements>
```

Within the `<execution:allocations>` element any number of `<execution:request>` elements can be provided describing individual requests for resources. For each `<execution:request>` a unique identifier must be provided in the attribute `id`. The `<execution:job>` block is used to describe the characteristics of the job to be conducted by the resource. A request message is created based on the key-value pairs specified through the `<execution:content>` elements provided for each job.

For each *Release Activity* a list of resource allocations to be released must be provided. The requested allocations to be released can be added to as shown below.

```
<extensionElements>
  <execution:allocations>
    <execution:release request="a_request_id"/>
    <execution:release request="another_request_id" />
  </execution:allocations>
</extensionElements>
```

Within the `<execution:allocations>` element any number of `<execution:release>` elements can be provided. Each `<execution:release>` must provide the identifier of the request to be released in the attribute `request`.

3 EXECUTION AND SIMULATION FRAMEWORK

This section describes a framework allowing to execute, simulate, and optimize process instances for resource-aware business process models using *Resource Activities*, *Request Activities*, and *Release Activities*, as well as status attributes, operators, and restrictions as described above. The proposed architecture is based on the observer design pattern and illustrated as an UML component diagram in Figure 4.

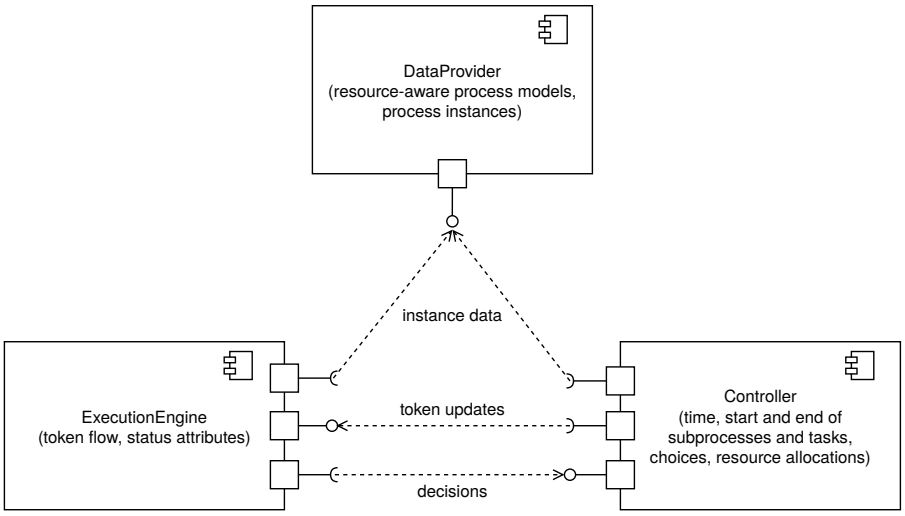


Figure 4: Core components of the framework.

The data provider reads all process models and instance data. It allows observers to subscribe to updates. Whenever new or modified instance data becomes available, all observers are notified allowing them to update their data accordingly.

The execution engine subscribes to the data provider to receive information about all process instances. Furthermore, the execution engine subscribes to a controller to receive the current time of the execution as well as all other decisions that must be made throughout process execution. Whenever the execution engine receives a new time or decision from the controller, the provided information is implemented accordingly and the execution engine advances process execution as far as it can on its own. The execution engine allows observers to subscribe to updates concerning the token flow. Whenever the state of the token changes or status attributes are modified, all observers are notified allowing them to react accordingly. Decisions that may have to be taken during process execution, i.e., decisions on whether to advance in time, whether to start or end a subprocess or task, which choice to be taken and which resource to allocate to a request, are not made by the execution engine.

Instead, the controller decides whether to advance in time or whether to take a decision that is necessary for the execution engine to proceed with process execution. The controller subscribes to the data provider to receive information about all process instances and the execution engine to receive information about the progress of process execution. Based on this information, the controller makes the decisions by any reasonable approach. Different controllers can be developed making decisions in different ways. For example, a controller can be developed that provides a dashboard for a human decision maker showing all necessary information. Alternatively, a controller can be

developed that applies appropriate rules for certain decisions that have to be taken, or a controller can make decisions using an appropriate optimization technique. Any controller allows observers to subscribe to all decisions made by the controller. Whenever a decision is made, all observers are notified allowing them to react accordingly.

The observer design pattern makes it easy to exchange specific implementations of the core components. For example, different controllers can be developed for different use cases without having the need to adapt the execution engine or the data provider. Furthermore, the observer design pattern makes it easy to integrate further components subscribing to the core components, e.g., a logger or performance analyzer.

4 EXAMPLE USE CASE: SHARED TAXI SERVICES

This section provides an example use case showing how the proposed framework can be used to execute, simulate, and optimize shared taxi services modelled by resource-aware business process models.

Figure 5 illustrates the process providing a *Resource Activity* representing a taxi. At the end of the process the *Return to depot* task ensures that each taxi eventually returns to the depot.

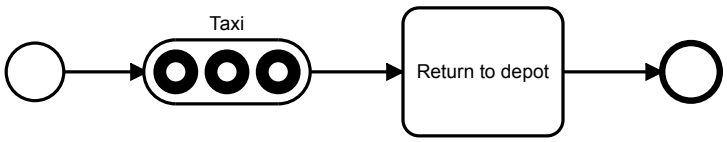


Figure 5: Process providing the taxi resource.

For each process instance, the data provider initializes the following status attributes: the attribute **time** gives the time at which the taxi begins service, the attribute **seats** gives the number of passenger seats of the taxi, the attribute **speed** gives the average speed of the taxi, the attribute **fare** gives the distance-based fare of the taxi, the attribute **cleaning_duration** gives the duration required to desinfect a used seat, the attributes **depot_x** and **depot_y** give the coordinates of the taxi depot, and the attribute **latest_allocation** gives the time when the taxi stops to offer the service.

When the process is initiated, the execution engine determines the current location of the taxi (stored in status attributes **current_x** and **current_y**) and the number of free seats (stored in status attribute **free_seats**) by applying the following expression operators.

```
<execution:operators>
  <execution:operator attribute="current_x">
    <execution:parameter name="expression" value="depot_x" />
  </execution:operator>
  <execution:operator attribute="current_y">
    <execution:parameter name="expression" value="depot_y" />
  </execution:operator>
  <execution:operator attribute="free_seats">
    <execution:parameter name="expression" value="seats" />
  </execution:operator>
</execution:operators>
```

A restriction ensures that the value of the attribute **free_seats** never falls below zero.

```
<execution:restrictions>
  <execution:restriction attribute="free_seats">
    <execution:minInclusive value="0" />
  </execution:restriction>
</execution:restrictions>
```

The taxi resource accepts request messages that provide the status attributes **pickup_x**, **pickup_y**, **dropoff_x**, **dropoff_y**, and **passengers** to be used with the *Allocation* event-subprocess illustrated in Figure 3 which are specified for the *Resource Activity* as follows.

```
<extensionElements>
  <execution:job>
    <execution:content key="PickupX" attribute="pickup_x" />
    <execution:content key="PickupY" attribute="pickup_Y" />
    <execution:content key="DropoffX" attribute="dropoff_x" />
    <execution:content key="DropoffY" attribute="dropoff_Y" />
    <execution:content key="Passengers" attribute="passengers" />
  </execution:job>
</extensionElements>
```

A status attribute **distance** with `objective="minimize"` and `weight="1"` indicates the Manhattan distance for the return trip from the current taxi location to the depot location. When conducting the *Return to depot* task, the distance, arrival time at the depot location, and the location of the taxi are determined using the following expression operators.

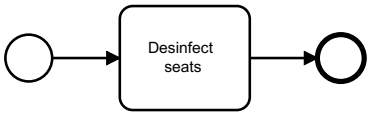
```
<execution:operators>
  <execution:operator attribute="distance">
    <execution:parameter name="expression" value="abs(depot_x␣-␣current_x)␣+␣abs(depot_y␣-␣
    ↪ current_y)" />
  </execution:operator>
  <execution:operator attribute="time">
    <execution:parameter name="expression" value="time␣+␣distance/speed" />
  </execution:operator>
  <execution:operator attribute="current_x">
    <execution:parameter name="expression" value="depot_x" />
  </execution:operator>
  <execution:operator attribute="current_x">
    <execution:parameter name="expression" value="depot_y" />
  </execution:operator>
</execution:operators>
```

Figure 6: *Default* subprocess of the taxi resource.

The *Default* subprocess of the taxi resource is shown in Figure 6 and only includes a timer event which triggers at the time specified for the **latest_allocation**-attribute.

(a) *Prepare* subprocess

(b) *Service* subprocess



(c) *Finish* subprocess

Figure 7: *Prepare*, *Service*, and *Finish* subprocesses of the taxi resource.

The *Prepare*, *Service*, and *Finish* subprocesses are shown in Figure 7. The *Prepare* subprocess and the *Service* declare a status attribute **distance** with `objective="minimize"` and `weight="1"` indicating the Manhattan distance from the current taxi location to the pickup location and the drop-off location, respectively. Furthermore, the *Prepare* subprocess declares a status attribute **revenue**

with `objective="maximize"` and `weight="1"` indicating the revenue obtained for the request. When the *Prepare* subprocess is initiated, the following expression operators are applied by the execution engine.

```
<execution:operators>
  <execution:operator attribute="free_seats">
    <execution:parameter name="expression" value="free_seats␣-␣passengers" />
  </execution:operator>
  <execution:operator attribute="distance">
    <execution:parameter name="expression" value="abs(pickup_x␣-␣current_x)␣+␣abs(pickup_y␣-␣
    ↪ current_y)" />
  </execution:operator>
  <execution:operator attribute="revenue">
    <execution:parameter name="expression" value="fare*(abs(dropoff_x␣-␣pickup_x)␣+␣abs(dropoff_y␣-␣
    ↪ pickup_y))" />
  </execution:operator>
</execution:operators>
```

It must be noted that it is the responsibility of the controller to ensure that all restrictions are satisfied when making decisions, in particular, those that impact the outcome of operators that change restricted attributes like the **free_seats**-attribute in this example.

When the *Drive to pickup location* task is executed, the arrival time at the pickup location, and the new location of the taxi are determined by the following expression operators.

```
<execution:operators>
  <execution:operator attribute="time">
    <execution:parameter name="expression" value="time␣+␣distance/speed" />
  </execution:operator>
  <execution:operator attribute="current_x">
    <execution:parameter name="expression" value="pickup_x" />
  </execution:operator>
  <execution:operator attribute="current_x">
    <execution:parameter name="expression" value="pickup_y" />
  </execution:operator>
</execution:operators>
```

When the *Service* subprocess is conducted, the distance, arrival time at the drop-off location, and the location of the taxi are determined using the following expression operators.

```
<execution:operators>
  <execution:operator attribute="distance">
    <execution:parameter name="expression" value="abs(dropoff_x␣-␣current_x)␣+␣abs(dropoff_y␣-␣
    ↪ current_y)" />
  </execution:operator>
  <execution:operator attribute="time">
    <execution:parameter name="expression" value="time␣+␣distance/speed" />
  </execution:operator>
  <execution:operator attribute="current_x">
    <execution:parameter name="expression" value="dropoff_x" />
  </execution:operator>
  <execution:operator attribute="current_x">
    <execution:parameter name="expression" value="dropoff_y" />
  </execution:operator>
</execution:operators>
```

When the *Finish* subprocess is initiated, the time when all used seats are desinfected and the number of free seats are determined using the following expression operators.

```
<execution:operators>
```

```
  <execution:operator attribute="time">
    <execution:parameter name="expression" value="time␣+␣cleaning_duration*passengers" />
  </execution:operator>
  <execution:operator attribute="free_seats">
    <execution:parameter name="expression" value="free_seats␣+␣passengers" />
  </execution:operator>
</execution:operators>
```

After the *Finish* subprocess is completed, the used seats are available again and can be used by other requests.

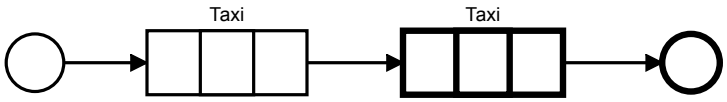


Figure 8: Process requesting a taxi.

Figure 8 shows a simple process requesting a taxi. The process begins with a *Request Activity* requesting the transport of passengers from a pickup location to a drop-off location. Whenever a suitable resource is allocated to the request, a request message containing the following information is sent to the resource.

```
<extensionElements>
  <execution:allocations>
    <execution:request>
      <execution:job>
        <execution:content key="PickupX" attribute="pickup_x" />
        <execution:content key="PickupY" attribute="pickup_Y" />
        <execution:content key="DropoffX" attribute="dropoff_x" />
        <execution:content key="DropoffY" attribute="dropoff_Y" />
        <execution:content key="Passengers" attribute="passengers" />
      </execution:job>
    </execution:request>
  </execution:allocations>
</extensionElements>
```

The *Request Activity* is completed when a taxi is available to pick up the passengers. Thereafter, the resource is released. The *Release Activity*, however, is only completed when the allocated taxi has completed its *Service* subprocess, i.e., has reached the drop-off location.

The data provider dynamically generates new process instances of this process with the status attributes required for the request: the attributes **pickup_x** and **pickup_y** give the origin coordinates of the passengers, the attributes **dropoff_x** and **dropoff_y** give the destination coordinates of the passengers, and the attribute **passengers** gives the number of passengers.

The allocation of requests to taxis as well as the sequence in which the respective trips are conducted is done by the controller and different controllers can be used ranging from manual controllers, over heuristic controllers subsequently selecting the nearest destination for the next trip, to sophisticated optimization methods exploiting information about the status of all process instances and the contribution of each decision to the objective.

The proposed framework can be used for both execution and simulation of these resource-aware business process models. For execution purposes, a data provider would give all information related to all taxi requests and would update status attributes, when necessary, e.g. if traffic conditions cause a delay. For simulation purposes, a data provider would be used that generates taxi requests according to some random distribution. Also status attributes could be updated dynamically, to simulate changing traffic conditions that can cause delays.

Various alternative use cases have been developed for the proposed framework ranging from inventory systems with periodic review to complex shipping processes in which multiple shipments are consolidated and transport resources themselves require resources such as drivers and vehicles. Due to the page limits, however, a detailed description of these use cases cannot be included in this paper.

5 CONCLUSION

This paper proposes a unifying framework allowing to model resource-aware business processes in such a way that they can be used for execution and simulation. The framework consists of three core components: a data provider, an execution engine, and a controller. The data provider and controller can be easily replaced depending on the use case and the entire execution logic is encapsulated within the execution engine. The framework is designed in such a way that it allows any decision mechanism to be deployed and that it allows for sophisticated optimization algorithms to be used. At the time of writing this paper a controller implementing a greedy one-step lookahead decision mechanism is developed that can be used to heuristically determine decisions that aim at optimizing the stated objectives. The development of controllers based on efficient optimization techniques, e.g. based on mixed-integer programming or deep reinforcement learning, is currently being investigated and an exciting direction for further research.

REFERENCES

Goel, A., and M.-B. Lin. 2022. "Resource Requirements in Business Process Modelling from an Operations Management Perspective". In *Proceedings of the 24th IEEE International Conference on Business Informatics*, 41–48: IEEE.

Gordon, G. . 1961. "A general purpose systems simulation program". In *AFIPS '61: Proceedings of the Eastern Joint Computer Conference*: Association for Computing Machinery.

Object Management Group 2013. "Business Process Model and Notation (BPMN) 2.0.2".

Onggo, B. S. S., N. Proudlove, S. D'Ambrogio, A. Calabrese, S. Bisogno, and N. Levialdi Ghiron. 2018. "A BPMN extension to support discrete-event simulation for healthcare applications: an explicit representation of queues, attributes and data-driven decision points". *Journal of the Operational Research Society* 69(5):788–802.

Pufahl, L., and M. Weske. 2013. "Batch activities in process modeling and execution". In *International Conference on Service-Oriented Computing*, 283–297. Springer.

Pufahl, L., T. Y. Wong, and M. Weske. 2017. "Design of an extensible BPMN process simulator". In *International conference on business process management*, 782–795. Springer.

Wagner, G. 2020. "Business Process Modeling and Simulation with DPMN: Resource-Constrained Activities". In *Proceedings of the 2020 Winter Simulation Conference*, edited by K.-H. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing.

Wagner, G. 2021. "Business Process Modeling and Simulation with DPMN: Processing Activities". In *Proceedings of the 2021 Winter Simulation Conference*, edited by S. Kim, B. Feng, K. Smith, S. Masoud, Z. Zheng, C. Szabo, and M. Loper.

Workflow Management Coalition 2013. "BPSim - Business Process Simulation Specification". Document Number WFMC-BPSWG-2012-1.

AUTHOR BIOGRAPHIES

ASVIN GOEL is Professor of Supply Chain Management and Logistics at Kühne Logistics University in Hamburg, Germany. Prof. Goel holds academic degrees from the Faculty of Mathematics at the University of Göttingen (Dipl.-Math.), from the Faculty of Mathematics and Computer Science at the University of Leipzig (Dr. rer. nat.), and from the Faculty of Law, Economics, and Business at the University of Halle-Wittenberg (Dr. rer. pol. habil.). His e-mail address is asvin.goel@the-klu.org. The modeller used for the examples shown in this paper is available at https://bpmn.telematique.eu.